Léon Valabrègue
Tristan Marandin

UFRJ
UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

Advanced programming: Project 1

Código da matéria: EEL418

Professor Cláudio Miceli
Matéria EEL418

Léon Valabrègue
Tristan Marandin

UFRJ
UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

I. INTRODUCTION

This report is part of the rendering of the project of Advanced Programming module, led by Cláudio Miceli here at the UFRJ. It is complemented by the following deliverables:

- A GitHub repository containing the source code: https://github.com/leonvala06/xoxo/
- A docker image of the working application: https://hub.docker.com/r/leonvala06/xoxo
  Create and start containers with the “docker-compose up” command. Then, open 2 browser windows and connect to localhost:3000. You are ready to play.

This application is a board game, tic-tac-toe or XOXO as commonly named. It is a two game player, the first to align 3 symbols wins.

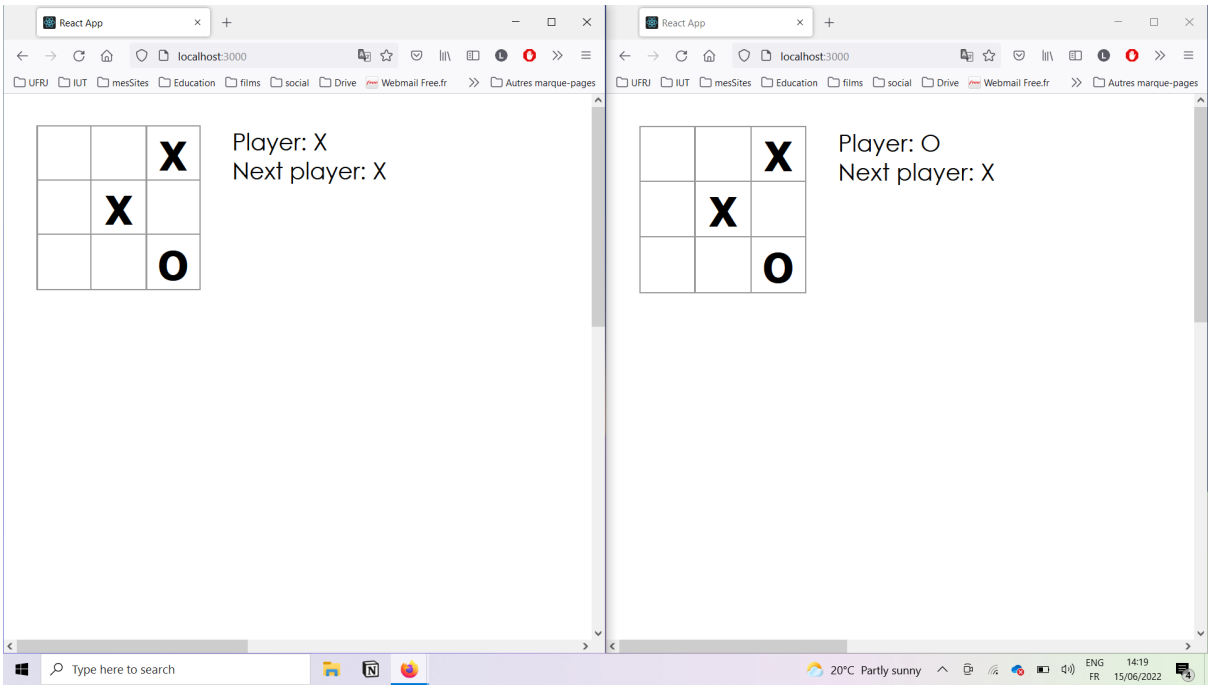

Fig 1: Running instance of the application

This app is a full stack web application using a MERN stack (Mongo, Express, React e Node.js). The details of this architecture will be further discussed in the report.

II. ARCHITECTURE OF THE APPLICATION

1. MVC or Microservices Architecture?

A. Definitions

First of all, we will begin by reminding the definitions of both the concepts of MVC and Microservices architecture.

MVC architecture

Stands for "Model-View-Controller." MVC is an application design model composed of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input).

The MVC model or "pattern" is commonly used for developing modern user interfaces. It provides the fundamental pieces for designing programs for desktop or mobile, as well as web applications. It works well with object-oriented programming, since the different models, views, and controllers can be treated as objects and reused within an application.

Below is a description of each aspect of MVC:

- the **Model** is data used by a program. This may be a database, file, or a simple object, such as an icon or a character in a video game.
- the **View** is the means of displaying objects within an application. Examples include displaying a window or buttons or text within a window. It includes anything that the user can see.
- the **Controller** updates both models and views. It accepts input and performs the corresponding update. For example, a controller can update a model by changing the attributes of a character in a video game. It may modify the view by displaying the updated character in the game.

The three parts of MVC are interconnected. The view displays the model for the user. The controller accepts user input and updates the model and view accordingly.

Microservices architecture

Microservices (or microservices architecture) are a cloud native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services. These services typically

- have their own technology stack, inclusive of the database and data management model;
- communicate with one another over a combination of REST APIs, event streaming, and message brokers; and
- are organized by business capability, with the line separating services often referred to as a bounded context.

B. MVC and microservices architectures assets

Then we will have to select one of the two previous architectures, one question remains : Which one is the most suitable to our project?

Why use MVC?

With the increasing complexity of systems/sites developed today, this architecture is focused on dividing one big problem into several smaller and less complex ones.

Thus, any kind of changes in any one of the layers does not interfere with others, making it easier to update layouts, change business rules, and add new features.

For large projects, MVC greatly facilitates the division of tasks among the team. Listed below are some of the benefits of using MVC:

- Easy maintenance and addition of features.
- Reduces project development time.
- Helps build reliable software with tested architectures.
- Facilitates code reuse.
- Greater team integration and division of tasks.
- Reduced complexity in code.

Why use Microservices?

Unlike traditional monolithic architecture, microservice architecture is an approach in which the application is structured through several small standalone services that work together. Today, with cloud platforms like AWS and AZURE, we can build Microservice architectures much more quickly, with native services that are self-managed or Serverless, for example, EC2, ALB, ECS, EKS, FARGATE, and several others.

Listed below are some of the benefits of using microservices :

- Makes continuous and automated deployment possible.
- Allows developers to make appropriate and service-specific decisions.
- Each microservice has its own database.
- Enables companies to optimize resources for development and applications.
- Optimizes sizing and easy to integrate with third-party services.

Considering the precedent points, the MVC architecture seems to be more suitable for our project, as a simple board game.

2. The View

On one hand, the view allows the user to play by displaying the board and let the player select both his pattern (‘X’ or ‘O’) and the cases he will mark.

On the other hand, the view communicates with the controller by two roads :

- GET : The view has to update every 500 ms to allow the user to stay aware of his opponent's moves.
- POST : The view sends the move of both the players to the controller to launch the process associated with the validation of the move and the update of the board.

3. The Controller

The controller stands between the model and the view. It is mainly composed of the server and the application express which is called by the first one.

The document *server.js* contains code lines to select the correct port on which he will run and some methods to facilitate the handling of errors. Finally, it listens to the events caused by user clicks on the view, and the application is supposed to handle them.

The document *app.js* contains :

- code lines to connect to the database.

```
7   // URL of the local database
8   const url = "mongodb://localhost:27017/"
9
10  // database connexion
11  mongoose.connect(url,
12  {
13    useNewUrlParser: true,
14    useUnifiedTopology: true,
15  });
16
17  const db = mongoose.connection;
18  db.on("error", (error) => console.error(error));
19  db.once("open", () => console.log("Connected to the database !"));
```

Fig 2: Connexion to the database

- code lines in charge of avoiding connection problems which the user could have for security causes.
- code lines to initialize the game, that’s to say to drop the precedent collection in the database and to create a new clean board to save.

```
// Let's drop the collection
db.dropCollection(
  "board",
  function(err, result) {
  console.log("Collection droped");
  }
  );

// Initialisation
const firstBoard = new Board({
  squares: Array(9).fill(null),
  turn: 0,
  attribute: 'X',
  issue: null
});

firstBoard.save()
.catch(error => console.log(error));

let turn = firstBoard.turn;
let attribute = firstBoard.attribute;
let issue = firstBoard.issue;
let idBoard = firstBoard._id;
```

Fig 3: Initialisation of the game

- a road GET, when the controller receives the update asking from the view, it collects the last board in the database to send it to the view, which will let it appear on both the users screens.

```
app.get("/getupdate", (req, res, next) => {
  Board.findOne({ _id: idBoard })
    .then(board => {
      let xIs = board.player == 'X' ? true : false;
      let xIsNext = nextPlayer(xIs);
      let newSquares = board.squares;
      let newIssue = board.issue;

      res.json({
        newSquares: newSquares,
        xIsNext: xIsNext,
        issue: newIssue
      })
    })
    .catch(err => console.log('get error : ' + err));
});
```

Fig 4 : GET road to refresh the view

- a road POST, when the controller receives a move from the view, it will check the validity of this move, and only if it passes the verification process, it will create a new board, from the precedent, and save it in the database.

```
167  app.post('/', (req, res, next) => {
168
169    // Check this turn
170    let isMyTurn = isMyTurnVerification(attribute, req.body.user);
171    let isNotOver = isNotOverVerification(issue);
172
173    if (isMyTurn == true && isNotOver == true) {
174      // Save this turn
175      turn += 1;
176      const newBoard = updateBoard(req.body.squares, turn, req.body.selectedSquare, req.body.user);
177      const finalBoard = gameIssue(newBoard);
178      saveBoard(finalBoard);
179
180      // Organise the next turn
181      attribute = getAttribute(nextPlayer(getPlayer(finalBoard.attribute)));
182      issue = finalBoard.issue;
183      idBoard = newBoard._id;
184    }
185  });
```

Fig 5 : POST road to update the database

- a road GET, which renders the database.

4. The Model

The model, in the backend, is a MongoDB database. This will store the different turns and the player's moves. This is running on the port 27017, and the controller will access it every time the view sends him a request.

- GET : The model sends to the controller the last board which has been saved. That occurs every second.
- POST : The model saves the new board sent by the controller after validation and update which is associated with a new turn, a new move.
- DELETE : The model erases the collection in which the different turns of the games are stored. That occurs at the beginning of a new game, to avoid the overcrowding of the database.

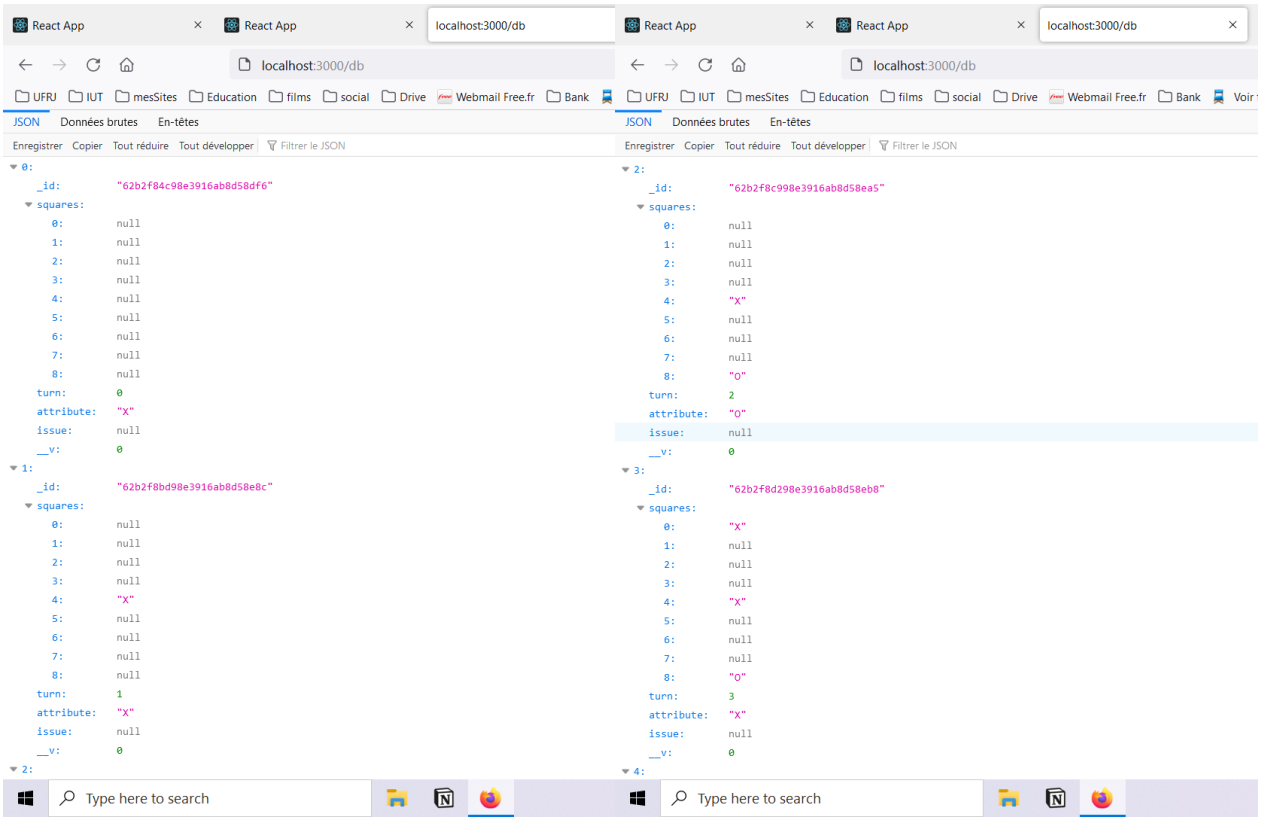

Fig 6: State of the database while game running

SATISFACTION OF THE REQUIREMENTS

The two main requirements were to use a MERN stack and use a MVC design pattern. These two were discussed in the previous section. In this section we'll talk about how CLEAN architecture and SOLID principles were applied during the development stage.

1. SOLID Principles

```
class Board extends React.Component {
  renderSquare(i) {
    return (
      <Square
        value={this.props.squares[i]}
        onClick={() => this.props.onClick(i)}
      />
    );
  }

  render() {
    return (
      <div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
          {this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}
          {this.renderSquare(7)}
          {this.renderSquare(8)}
        </div>
      </div>
    );
  }
}
```

Fig 7: Extract of index.js file (client)

- Single Responsibility Principle

We tried as much as possible to assign one responsibility for one class. For instance, the Board class has one responsibility: holding the board that is composed of 9 squares.

- Open Closed Principle

We used inheritance and composition in the client. We built encapsulated React components that manage their own state, and then we composed them to make the user interface. As a result, the *Game* class is composed of the *Board* class, which itself is composed of *Square* elements.

- Liskov Substitution Principle

We don't have functions that use pointers or references to base classes so we weren't concerned about applying this principle.

- Interface Segregation Principle

We have small and cohesive interfaces to implement if changes are made in the client or database system.

Regarding the client: two functions are primordial to communicate with controller:

```
-    getUpdate(leGame)
-    sendUpdate(newSquares, i)
```

Fig 8: extract of index.js file of client

Regarding the database, if we change it, it will be enough to redefine the GET, POST and DELETE procedures in the controller, defined in file *app.js*.

- Dependency Inversion Principle

We didn't have to build a class which had dependencies on other classes or interfaces, so we weren't concerned by this situation.

2. CLEAN Architecture

We applied the CLEAN architecture to our interactions with the Database. The application is built around an independent object model that can be run separate from it.

```
PS C:\Users\leloune\OneDrive - IMT MINES ALES\Documents\MINES\S8\UFRJ\ProgAv\alattaque> docker-compose up
Starting mongo ... done
Recreating app ... done
Attaching to mongo, app
```

Fig 9: Creating and starting 2 containers: app for the application and mongo for the database.

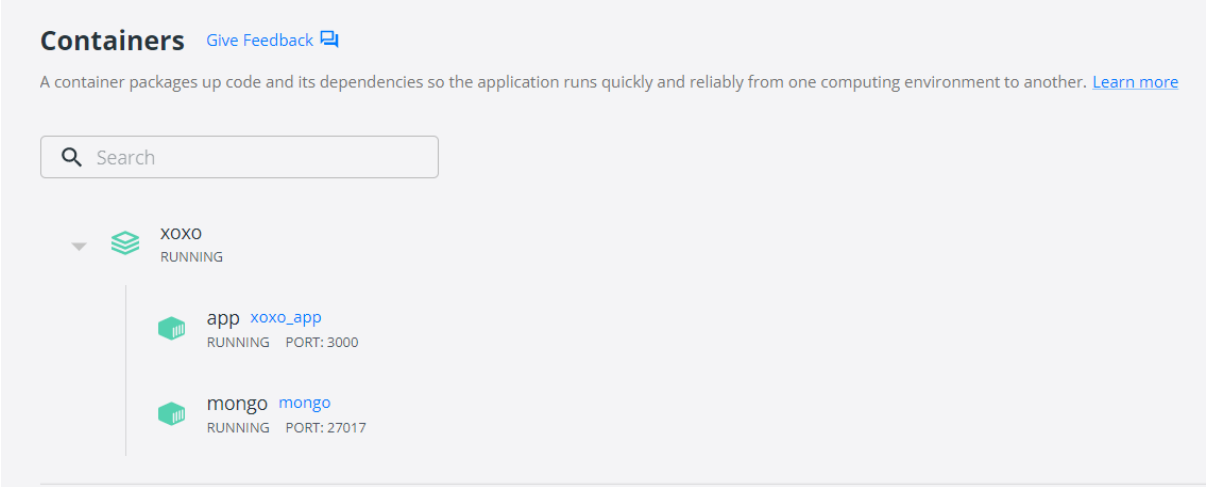

Fig 10: Visualization in Docker Desktop of the two independant Containers running

III. POSSIBLE IMPROVEMENTS

I. Frontend Design

If you've run the program, you should have noticed that the user interface is minimalist. It's extremely simple, and functional. However, some web design sessions could be achieved to make the interface more attractive.

II. Middle End Optimisation

The controller is composed of a document *Server*, an *Express Application*, and a folder *Model* which contains the mongoose scheme of the board. To optimize this part, it could be relevant to separate the road in a single document *Router*, roads which will call functions, set in another document *Controllers*.

This modification allows a code more readable, which is important if other people have to work on it, to develop a new functionality.

III. New Game

If you want to start a new game, you will have to restart the server. To add a functionality which will allow both of the players to play again by clicking on a simple button, is quite relevant. In order to achieve it, you could write a function initialisation, called by a new road POST, which is triggered by clicking on the matching button.

To implement this new functionality, you will have to deal with the asynchronism problem.

IV. SOURCES

- Intro to React Tutorial on the official website. We used the tic-tac-toe example used in the tutorial as a base of our client side application. https://reactjs.org/
- Documentation MVC and Microservices architecture. We used this documentation to help us make and justify a choice of architecture. wisdomplexus.com
- SOLID Principles. We used this website to follow the common coding principles known as SOLID. https://davesquared.net/2009/01/introduction-to-solid-principles-of-oo.html